[redacted]

Report No. UIUCDCS-R-75-790

NSF 21590 A02

CAPS Compiler CPU use Report

by

Lawrence A. White

December 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-75-790

CAPS Compiler CPU use Report

by

Lawrence A. White

December 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. NSF 21590 A02.

Table of Contents

Abstract

This report describes an investigation into the internal workings of the driver for the CAPS compilers, and discusses the results of this investigation. The primary questions were where and how was CPU time being used in the compilers, and how could their CPU consumption be reduced to speed them up. The results, as described in this report, show a possibility for a minimal improvement, say 10%, by recoding critical sections of the compiler driver. Further improvements appear feasible only by sacrificing some features of the compiler, or by reducing the generality of the compiler driver.

This report is the result of an investigation done as a CS425 project for Professor Thomas R. Wilcox at the University of Illinois. The investigator and author of the report, Lawrence A. White, is a Graduate Student at the University. Additionally, he is an experienced system programmer on the Plato computer system on which the CAPS compilers run, has his own compiler system on Plato, and is quite qualified to conduct the investigation described in the following report.

Section 1: Description of Compiler Environment

The CAPS compilers are implemented on the Plato Computer System, a computer-based educational system developed at the University of Illinois. This system is designed to run up to 500 students simultaneously in many instructional areas. Most students are in areas other than Computer Science, and it is still to be determined whether Plato can be used effectively to teach a subject that may require a high CPU activity, such as programming. This section of the report describes the portions of the Plato system relevant to the CAPS compilers.

1.1: Tutor Lesson Organization

Programs on Plato are organized into "lessons", each consisting of up to eight or ten thousand words of pure procedure, shared by any number of students at once. These lessons are "condensed", rather than compiled, into portions of interpretable and directly executable code. Data areas referenced by these lessons are described in section 1.2.

The only language supported by Plato for its users is TUTOR, which has been developed along with the rest of the Plato system. To allow students to program in another language, a compiler and interpreter must be written in Tutor. The CAPS compilers are written in Tutor, and suffer the advantages and disadvantages thereof.

All information needed for each active student is kept in ECS (Control Data Corporation's Extended Core Storage) during a

session. This includes each student's current lesson and all data areas referenced by that lesson. When the CPU is actually working on an individual student, portions of his lesson and data areas are swapped into CM (Central Memory) for processing, and stored back in ECS between timeslices.

1.2: Tutor Data Areas

The CAPS compilers may access three data areas in ECS for running students. The first area, called "student variables", is individual to each student, is 150 words long, and is loaded and unloaded from ECS to *n* variables in CM every timeslice. The second data area for each user, "storage", may be up to 1500 words long, and portions of it and "common", the third area, will be loaded and unloaded into 1500 words of *nc* variables in CM as directed by the lesson. Storage is individual to each user of the lesson, but unlike student variables, storage is not saved on disk between sessions. Common is shared by all users of a lesson, may be up to 8050 words long, and may reside on disk while no user is referencing it.

Section 2: CAPS Compiler Organization

Each compiler in the CAPS system consists of interpretive tables specific to the language being compiled; common driving routines to interpret these tables; and a few routines, specific to the language, that are called from the interpreted tables. These tables are built from assembler-like source code written by a compiler implementor. After generation, these tables are stored in common where they are loaded into *nc* variables as needed in compiling student programs.

2.1: Compiler Modules

Flow of control in the CAPS compilers is shown in Figure 1. The editor looks at each keypress the student enters from the terminal. If the key indicates a text editing function, it is performed by the editor. If the student is entering new text, each keypress is passed on to the lexical analyzer. When the lexical analyzer receives a complete token, that token is passed on to the syntax analyzer and parser for compilation. Since each keypress is processed as it is entered, the compiler can give immediate error messages when the student enters an invalid language construct.

While compiling new text, "Trace" information is stored, allowing the Reverse Editor to uncompile the student's program as the student backs up to make a change. Occasionally the storage area for Trace information gets full. When this happens, a compression unit is called which removes alternate entries from the Trace table. After compression is performed,

the reverse editor can only back up to alternate tokens. If necessary, it will back up to the previous token and then forward compile to the current token. In practice, the compression routine may be called three or four times for a student program. After four calls, there is Trace information for one out of every 16 of the first tokens entered. Closer to the "cursor" where the student is working, the Trace information is available for every token, or at least alternate tokens.

The lexical analyzer and the parser are both table driven. The table for the lexical analyzer is a state transition diagram interpreted by Tutor code. For example, if currently in the '<' node of the diagram for PL/1, a following '=' causes transition to another node, while a following '<' or '>' is an error noted by the lexical analyzer. While in BASIC, for example, '<>' is the "not equal" token.

The tables in the Parser are not just a state transition array, as in the lexical analyzer, but consist of internal codes interpreted by a Tutor unit in each compiler. This internal code is complete with arithmetic operations, conditional jumps, calls to error routines, and calls to the lexical analyzer to receive the next token. Thus, the student writes in PL/1 for example, and his program is compiled by code being interpreted by Tutor code being interpreted by Plato run time routines.

The module labeled "syntax analyzer" in Figure 1 is just an interface between the lexical analyzer and the parser. Its function is to keep track of trace information and to insure that the correct tables are loaded for each routine.

```
 r---------¬        r---------¬        r---------¬
 |         |        |         |        |         |
 | Student |------->|  Editor |------->| Reverse |
 | Terminal|<-------|         |<-------|  Editor |
 |         |        |         |        |         |
 L---------J        L---------J        L---------J
                         |
                         |
                         |
                         |
                         |
                         V
                    r---------¬
                    |         |
                    | Lexical |=======> Name Table
                    | Analyzer|
                    |         |
                    L---------J
                         |
                         |
                         |
                         |
                         |
                         V
 r---------¬        r---------¬
 |         |        |         |
 | Compress|<-------|  Syntax |=======> Trace
 | Module  |------->| Analyzer|
 |         |        |         |
 L---------J        L---------J
                         |
                         |
                         |
                         |
                         |
                         V
                    r---------¬
                    |         |
                    |  Parser |=======> Symbol Table
                    |         |=======> Trace
                    |         |
                    L---------J
```

Figure 1: CAPS Compiler Modules

2.2: Compiler Data Structures

The CAPS compilers use "common" for pointers and tables shared by all users of one compiler and "storage" for all pointers and tables needed by an individual user. Few, if any, of the student variables are used by the compilers. Portions of common and storage may be loaded into 1500 *nc* variables in CM. However, at most three areas of each may be loaded at once. As shown in Figure 2, by arranging the data areas in ECS carefully, it was possible to meet this three area restriction and still get the tables in desired locations in CM. However, the lexical and parse tables are each 400 words long, and only one of them could be loaded at once. (As shown later, this is significant since the compilers spend 8% of their time changing the loading arrangement.) Figure 2, showing the layout of these areas, follows on the next page.

```
         CM                              ECS

 nc variables (1500)              Storage         (644)
┌──────────────────────┐         ┌──────────────────────┐
|                      |         |   Parse Storage 53   |
|   Lexical            |         ├──────────────────────┤
|     or               |         |v  Symbol Table 109   |
|   Parse              |         ├──────────────────────┤
|   Tables             |         |v  Name Table    64   |
|                      |         ├──────────────────────┤
|   400                |         |v  Char Table   168   |
|                      |         |                      |
├──────────────────────┤         ├──────────────────────┤
|   Parse Storage 53   |         |   Hash Table    20   |
├──────────────────────┤         ├──────────────────────┤
|v  Symbol Table 109   |         |   Text          60   |
├──────────────────────┤         ├──────────────────────┤
|c  Symbol Table 210   |         |   Trace         98   |
|                      |         ├──────────────────────┤
├──────────────────────┤         |   Variables     88   |
|c  Name Table    110  |         └──────────────────────┘
├──────────────────────┤
|v  Name Table     64  |         Common          (1288)
├──────────────────────┤         ┌──────────────────────┐
|v  Char Table    168  |         |                      |
|                      |         |   Parse              |
├──────────────────────┤         |   Table              |
|c  Char Table    119  |         |                      |
|                      |         |   400                |
├──────────────────────┤         |                      |
|   Hash Table     20  |         |                      |
├──────────────────────┤         ├──────────────────────┤
|   Text           60  |         |                      |
├──────────────────────┤         |   Lexical            |
|   Trace          98  |         |   Table              |
├──────────────────────┤         |                      |
|   Variables      88  |         |   400                |
└──────────────────────┘         |                      |
                                 ├──────────────────────┤
                                 |   Pointers      22   |
                                 ├──────────────────────┤
 v = variable portion            |c  Symbol Table 210   |
 c = constant portion            |                      |
 number = length of table        |                      |
                                 ├──────────────────────┤
                                 |c  Name Table   110   |
                                 ├──────────────────────┤
                                 |c  Char Table   119   |
                                 └──────────────────────┘
```

Figure 2: CAPS Data Areas

Section 3: Timing Measurements

The object of these timing tests was to determine where CPU time was used while entering text. Additional tests may later be run to determine CPU use when backing over or reparsing already entered text. Each of these tests involved entering the following sample PL/1 program a number of times and taking the average or most frequent timing result. Most of the timing tests were done in the middle of the day during the week, the period of highest Plato use, and the results were fairly consistent.

```
P:        PROC;
          DCL A(3,3) FLOAT, (I,J) FIXED;
          DO I=1 TO 3;
             DO J=1 to 3;
                A(I,J)=I+J;
                END;
             END;
          END;
```

In May 1975, Al Davis used this program in some timing tests. His results showed CPU times ranging from 1.7 to 1.9 CPU seconds. The author received similar results when he started his tests, as average of 1.75 CPU seconds per compilation.

3.1: Command Statistics

An option is available to Plato system programmers to take command execution statistics for individual lessons. When the author turned on this option for the compiler, he received the following averages for a single compilation of the above

program. The commands shown took more than 1% of the CPU time used or the number of commands executed, and are broken down into various types.

Command	Count	%	Time	%	Ave Time
display - 19% total time					
at	433	6.26	46	8.48	.106
mode	318	4.60	24	1.82	.075
showa	182	2.63	44	3.33	.242
showt	89	1.29	31	2.35	.348
write	196	2.83	48	3.63	.244
calculations - 51% total time					
calc	1789	25.85	609	46.14	.340
calcs	333	4.81	72	5.45	.216
program control - 20% total time					
arg	450	6.50	36	2.73	.080
do	1004	14.51	111	8.41	.110
entry	172	2.49	15	1.14	.087
goto	150	2.17	17	1.29	.113
gotoc	277	4.00	17	1.29	.061
jcinc	85	1.23	12	.91	.141
unit	592	8.55	55	4.17	.093
other - 10% total time					
pause	90	1.30	35	2.65	.389
comload	140	2.02	75	5.68	.536
stcload	53	.77	26	1.97	.491

Count = number of times command was executed. Times are shown in milliseconds. The "arg" command is generated by the Plato condensor for picking up parameters passed to units.

The reader may note that the sum of the times shown is less than the 1.75 CPU seconds previously mentioned. In fact, command statistics increased the CPU time by 20% to 2.10 seconds, but the statistics do not take into account formatting time of output to the terminal or the overhead of timeslicing, nor do they show all commands.

3.2: Module Statistics

As well as knowing what commands were using CPU time, the author wanted to know how much time was being spent in each module of the compiler. To learn this, he made a copy of the PL/1 compiler and modified it to record timing statistics on entry and exit from each module. This additional code increased compile time by 34% to 2.34 CPU seconds, and produced the following data. Times shown here are in seconds.

Module	CPU Time	Entry Count	% Time
Editor	.07	1	3
Lexi	.59	172	25
Syna	.33	51	14
Comp	.23	2	10
Parse	1.12	51	48

The data shows 172 characters (including spaces in TABS) entered and scanned by the lexical analyzer. The lexical analyzer recognized 51 tokens and passed them on to the syntax analyzer and parser, with the syntax analyzer having to call the compression routine twice.

Combining module statistics with the command statistics shown in section 3.1, we see that 14% of the total time goes into the syna module, with 8 of the 14% being used just to change the CM loading information for the parser and then back for the lexical analyzer. Parsing consumes the largest portion of CPU time, namely 48% of it, and involves interpreting of the internal code in the parse table.

3.3 Removing Trace Collection

Most of the Trace information generated is due to building the symbol table and the necessity of restoring it to previous states if the user backs over DECLARE statements. If semantic checks, including symbol table generation, were removed from the compiler, some improvement in speed should be expected. To test this hypothesis, the author removed all storing of Trace information and then ran the same data collection routines again. The amount of CPU time used was reduced by 20%, with command statistics and module timing shown below. (Times again are in milliseconds for command statistics, in seconds for module statistics.)

Command	Count	%	Time	%	Ave Time
display - 22% total time					
at	433	6.49	44	8.64	.102
mode	318	4.77	13	1.08	.041
showa	182	2.73	50	4.14	.275
showt	89	1.33	33	2.73	.371
write	196	2.94	71	5.87	.380
calculations - 47% total time					
calc	1710	25.63	509	42.10	.298
calcs	333	4.99	61	5.05	.183
program control - 19% total time					
arg	429	6.43	42	3.47	.098
do	931	13.97	115	9.51	.131
entry	172	2.58	13	1.08	.076
goto	150	2.17	8	.66	.053
gotoc	226	3.39	18	1.49	.080
joinc	85	1.27	4	.33	.047
unit	569	8.53	36	2.98	.063
other - 12% total time					
pause	88	1.32	29	2.40	.330
comload	140	2.10	88	7.28	.629
stoload	53	.79	40	3.31	.755

Module	CPU Time	Entry Count	% Time	% Original Time
Editor	.07	1	3.5	3
Lexi	.56	172	28	24
Syna	.30	51	15	13
Comp	0	0	0	0
Parse	1.08	51	53.5	46
				86%

The difference between the 20% improvement mentioned above and the 14% improvement shown by these statistics results from the statistics overhead not being reduced significantly by removing the Trace collection.

Section 4: CPU Time Improvements

One of the goals of this investigation was to determine what coding changes could be made in the compilers, and how much of a timing improvement could be expected. Currently a student may receive up to 10 milliseconds of CPU time per real time second. To enter this sample program under such conditions takes almost three minutes. To anyone who has used the system, it is apparent that an improvement is desired.

4.1 Possible Improvements

Four suggestions for improving CPU time have been made. The first involves minor recoding of critical sections of the compiler and would give a minor improvement in speed. Two such changes apparent to the author are to 1) stop displaying the "space left" indicator on the student's screen -- 2.5% improvement, and 2) stop doing unnecessary -stoload- commands -- 2% speedup. (The compiler was reexecuting the same -stoload- command once for every token, which is unnecessary, at least for the PL/1 compiler.)

The second improvement suggestion involves recoding the lexical analyzer or parser in Tutor, rather than having Tutor code interpret these tables. This would give an unknown amount of speedup, estimated by the author at about 20% for the lexical analyzer, more for the parser. However, this suggestion was not received too enthusiastically by the compiler designer since it reduced the generality of the driver program.

The third method proposed for speeding up the compilers is to collect a whole line of source text at a time before processing, rather than a character at a time. This moves the collection process from the Tutor lesson to the Plato system, and allows the user to correct errors in his current line before giving it to the lexical analyzer and parser. This method has been implemented and tested, and does reduce the CPU time used. However, this contradicts one of the basic goals of the CAPS system, namely that of responding immediately to an invalid character or token.

The fourth method suggested for improving compile time speed was to move semantic (symbol table) checking to a later pass. This method appeared to be promising enough that further tests were performed to determine how much speedup could be expected, with the results of those tests shown in section 3.3. The data in section 3.3 does not completely describe the improvement due to removing semantic checks, since some trace information would still have to be collected. Even so, the improvement is probably still more than the 20% shown in 3.3 since the parser continued to perform semantic checks while the data was collected.

4.2 Recommended Design Changes

The previous section described several independent improvements that have been suggested. This section describes a coherent set of changes, some suggested by the author, others by the CAPS designer, T. R. Wilcox. Here these suggestions are

combined together in a manner that will yield a clean compiler design with improved speed. They are not just restricted to the entry of new text, but cover the editor, compiler, and executor for student programs. The design is based on the timing statistics shown in previous sections and the author's experience in Plato compilers.

1) Move semantic checking to a new pass immediately preceding execution. This new pass should be coded in Tutor and be specific to the language being compiled.

2) In this new pass, compile the user's program into a more easily interpretable form than the tokenized input. This suggestion is dependent on the results of some experiments currently being performed, and is based on the author's experience with another compiler system that executes roughly four times faster. However, the resultant speed cannot match that of the other system since the CAPS run time routines must store more error detection and correction information.

3) Once semantic checking has been removed from the editor, no reparsing of source text needs to be done as the user moves the cursor either forward or backwards over previously entered text. Thus, once an expression has been parsed, it never needs to be parsed again unless the user changes something in it. This could be applied to subexpressions, but probably only "expressions" and "statements" need to be

recognized as units to give the desired speedup. The editor function keys might then become "intelligent" enough to recognize such commands as "back up to start of expression" or "move forward one statement", rather than just recognizing characters, words, and lines, as it currently does.

4) After 1, 2, and 3 above, this author knows of no way to improve the speed of the compiler other than recoding part or all of it in Tutor. A reasonable speed will probably not be obtained with the present Plato CPU limitations until the lexical analyzer, at least, is recoded in Tutor. Recoding the parser in Tutor would be easier after semantic checks are moved to a separate pass, but would still be a large job.

Suggestion 1 gives a speedup that is not nearly proportional to the amount of recoding involved. However, it allows suggestion 3, which gives an improvement greater than the amount of recoding needed to implement it. Once we reach suggestion 4, the amount of speedup has become directly proportional to the amount of recoding done.

In summary, this report describes the Plato system organization and the compiler design features relevant to the speed of the CAPS compilers. It shows where, and how, CPU time is being used, and it presents some ideas for speeding up the compilers. As the investigator in this report, I am impressed with the design and implementation of the compiler driver. It

appears to be well structured and clearly thought out, and a diagnostic compiler that runs in this environment on the order of real time is a significant addition to the field of Computer Science. In particular, this compiler appears to meet all of its design goals except speed, and this report shows ways of speeding it up. It has been shown possible to run non-diagnostic student compilers on Plato (see Appendix B), and this author believes that the CAPS compilers can be improved to run in real time, without loss of diagnostics for the student. However, this might require recoding everything in Tutor, as in the other compiler system.

Appendix A: Parser Internal Code Counts

While investigating the CPU use in the CAPS compiler, the author counted the number of each type of parser operation executed during compilation. Though this information was not used in the basic report, it is included here in case someone desires this data. Also, it might be used to make minor code changes in the parser interpreter to speed up the most frequent operations. These counts are derived from one entry of the sample PL/1 program given above.

Code	Count	Operation
0	51	Scan
1	6	Allocate
2	5	Deallocate
3	30	Call
4	30	Return
5	18	Semantic Op
6	20	Branch
0	66	= branch
1	61	≠
2	0	>
3	0	≥
4	0	<
5	0	≤
6	0	notall branch
7	7	notany
8	0	all
9	1	any
10	29	ps(X) = Y
11	23	= locY
12	0	= ps(X)$mask$Y
13	0	= ps(X)$union$-Y
14	0	= ps(X)+Y
15	0	= ps(X)-Y

Appendix B: Another Compiler System

Some readers might be interested in comparing CPU use for the CAPS compilers with another compiler system on Plato. Such comparisons will undoubtedly be unfair to the CAPS system since the goals of the two systems are very different. This other system has been developed by Axel T. Schreiner and the author, and currently contains compilers for subsets of BASIC and Fortran. The information here is incomplete, covering only Fortran compilation, while ignoring editing and execution. The sample program, shown below, is a simple numerical integration program of about 18 statements. Compilation of this program from source to an easily executable internal code took about .46 CPU seconds. The command statistics for one compilation appear following the program.

```
      FUNCTION F(X)
      F=X*X+4.
      RETURN
      END
C
10    PRINT,'INTEGRATE FROM'
      READ,A
      PRINT,'INTEGRATE TO'
      READ,B
      PRINT,'STEP SIZE'
      READ,STEP
      J=(B-A)/STEP-1.
      SUM=(F(A)+F(B))/2.
      DO 20 I=1,J
      SUM=SUM+F(A+I*STEP)
20    CONTINUE
      PRINT,'AREA',SUM*STEP
      GOTO 10
      END
```

Command	Count	%	Time	%	Ave Time
display - 10% total time					
at	77	2.15	3	.66	.038
erase	26	.73	4	.87	.154
mode	1	.03	-	-	-
showa	19	.53	8	1.75	.421
showt	41	1.15	23	5.02	.561
write	31	2.11	6	1.32	.517
calculations - 44% total time					
calc	369	24.32	136	29.69	.157
calcc	9	.25	-	-	-
calcs	19	.53	-	-	-
move	94	2.63	26	5.68	.277
find	103	2.88	24	5.24	.233
search	28	.78	12	2.62	.429
program control - 45% total time					
arg	244	6.83	17	3.71	.070
do	524	14.67	63	13.75	.081
entry	23	.64	5	1.09	.217
exit	50	1.40	5	1.09	.100
goto	98	2.74	7	1.53	.071
gotoc	931	26.06	85	18.56	.091
unit	309	8.65	24	5.24	.078
other					
comload	2	.06	-	-	-
stoload	1	.03	-	-	-

IOGRAPHIC DATA ET	1. Report No. UIUCDCS-R-75-790	2.	3. Recipient's Accession No.
le and Subtitle PS Compiler CPU use Report			5. Report Date December, 1975
			6.
thor(s) Lawrence A. White			8. Performing Organization Rept. No.
forming Organization Name and Address iversity of Illinois at Urbana-Champaign partment of Computer Science bana, IL 61801			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NSF 21590 A02.
ponsoring Organization Name and Address National Science Foundation Washington, D. C.			13. Type of Report & Period Covered
			14.

upplementary Notes

bstracts
is report describes an investigation into the internal workings of the driver for CAPS compilers, and discusses the results of this investigation. The primary stions were where and how was CPU time being used in the compilers, and how could ir CPU consumption be reduced to speed them up. The results, as described in this ort, show a possibility for a minimal improvement, say 10%, by recoding critical tions of the compiler driver. Further improvements appear feasible only by sacrific- some features of the compiler, or by reducing the generality of the compiler driver. is report is the result of an investigation done as a CS 425 project for Professor mas R. Wilcox at the University of Illinois. The investigator and author of the ort, Lawrence A. White, is a Graduate Student at the University. Additionally, he an experienced system programmer on the Plato computer system on which the CAPS pilers run, has his own compiler system on Plato, and is quite qualified to conduct investigation described in the following report.

ey Words and Document Analysis. 17a. Descriptors

Interactive compilers; performance measurements

PLATO, programming systems, text editors

dentifiers/Open-Ended Terms

OSATI Field/Group

ailability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 25
Release Unlimited	20. Security Class (This Page UNCLASSIFIED	22. Price